

System C#

C# for the systems minded programmer

Jared Parsons

Language History

- Midori
 - Ground up OS effort written in C#
 - Compiled to native with Bartok and Phoenix (x86, amd64 and arm)
 - Correct by construction, non blocking API calls
- Embraced rigorous static analysis and fail fast discipline
 - Invested heavily in FxCop to enforce rules (stack only, immutable types, exceptions)
 - Had over 100 custom rules
- Eventually reached the limits of FxCop
 - Too easy to suppress, generics were always a problem, hard to enable in old code
 - At their core the rules building a new type system
 - FxCop was simply not meant to do this

System C#

- Made the decision to invest in language changes
 - Build a real type system that is universally and ruthlessly enforced
 - Developers can't silence compiler errors with an attribute
- M# kicked off in 2010
 - Based on C# 4.0 (and eventually merged with 5.0)
 - Started with small impactful features (awaits, ref returns)
 - Graduated to much larger type system changes (immutability, contracts, error model, destruction, etc ...)
- M# renamed to System C# in 2014
 - It is not Redhawk's System C# (has similar feature sets)

System C# Philosophy

- System C# is to C# what C++ is to C
 - Evolution not revolution
- Design Principles
 - Do not break back compat with C# unless absolutely necessary
 - Features can be adopted incrementally
 - Prefer explicit over implicit
 - C++ code quality (speed and size) with C# productivity
- C# developers can code System C# today
 - Existing knowledge + patterns can be directly applied
 - Adopt features as needed
 - Back compat breaks are minor and controllable with compiler flags

Destructible and Borrowed

Adding explicit ownership to improve resource management

Motivation

```
throws await void Parse(Stream stream)
{
    Buffer<byte> buffer;
    try {
        while (try await stream.Peek(out buffer)) {
            Span<byte> span = buffer.Value;

            int bytesConsumed = try headerParser.Process(span);
            buffer.Dispose();
            try await stream.Skip(bytesConsumed);
            if (headerParser.Done) {
                break;
            }
        }
    }
    finally {
        buffer.Dispose();
    }
}
```

Motivation

```
throws await void Parse(Stream stream)
{
    Buffer<byte> buffer;
    try {
        while (try await stream.Peek(out buffer)) {
            Span<byte> span = buffer.Value;

            int bytesConsumed = try headerParser.Process(span);
            buffer.Dispose();
            try await stream.Skip(bytesConsumed);
            if (headerParser.Done) {
                break;
            }
        }
    }
    finally {
        buffer.Dispose();
    }
}
```

Is this disposable?

Do I dispose this?

Potential dangling pointer

Duplicate dispose code

Borrowed Values

- **Solution:** Encode ownership into the type system
- Represent “use but don’t store” pattern
 - Inspired by Rust borrowed pointers, C++ references
 - & type modifier applies to references, values and members
 - E.g., `List<Foo>&` is a borrowed list
- Provides definitive lifetime information for values
 - Methods can’t stash away borrowed parameters
 - Borrowed values can’t be returned beyond the scope they are allocated in
- Allows compiler to safely stack allocate objects
 - Not dependent on fragile escape analysis
 - Compiler knows creation point
 - No one can store the value past the creation point

Destructible Types

- Deterministic value destruction
 - Inspired by C++11 smart pointers
- Ownership is explicit
 - Owned<Foo>: single value, single owner
 - Shared<Foo>: single value, many owners
 - Foo&: underlying value can't be destructed while Foo& is in scope
- Works on classes and structs
- Destruction safety enforced at compile time
 - No dangling pointers

Ownership Becomes Obvious

```
throws await void Parse(Stream& stream)
{
    Shared<Buffer<byte>> buffer;
    while (try await stream.Peek(out buffer)) {
        readable byte[]& span = buffer.Value;

        int bytesConsumed = try headerParser.Process(span);
        try await stream.Skip(bytesConsumed);
        if (headerParser.Done) {
            break;
        }
    }
}
```

Ownership Becomes Obvious

```
throws await void Parse(Stream& stream)
{
    Shared<Buffer<byte>> buffer;
    while (try await stream.Peek(out buffer)) {
        readable byte[]& span = buffer.Value;

        int bytesConsumed = try headerParser.Process(span);
        try await stream.Skip(bytesConsumed);
        if (headerParser.Done) {
            break;
        }
    }
}
```

Explicitly not owned

Ownership Becomes Obvious

```
throws await void Parse(Stream& stream)
{
    Shared<Buffer<byte>> buffer;
    while (try await stream.Peek(out buffer)) {
        readable byte[]& span = buffer.Value;

        int bytesConsumed = try headerParser.Process(span);
        try await stream.Skip(bytesConsumed);
        if (headerParser.Done) {
            break;
        }
    }
}
```

Explicitly not owned

Explicit shared ownership

Explicit lifetime

Taming Side-Effects (TSE)

Immutability, Isolation, Side-Effects in the Type System

Motivation

- APIs today have implicit contracts about side-effect behavior
 - And callers make assumptions about these contracts
 - Examples of such contracts
 - Callee may read but not mutate parameters
 - Caller may not mutate values while callee is running
 - Method must be pure
- Unenforced, leading to brittle/buggy code
 - Allocate wrapper objects like `ReadOnlyCollection<T>`
 - Strongly worded comments
 - Particularly dangerous when concurrency is in the picture

The Basic Idea: Permissions

- **Solution:** Restrict permissions, add first class concept of (im)mutability
- References, values, and methods are annotated with permissions.

| | |
|-------------------------------------|--|
| <code>writable</code> Foo a = ...; | Object graph can be mutated (default if unstated) |
| <code>readable</code> Foo b = ...; | Object graph cannot be mutated (by me) |
| <code>immutable</code> Foo c = ...; | Object graph is frozen until the end of time |
| <code>P</code> Foo d = ...; | Generic permission (advanced; more later) |

- Modeled as subtypes; behavior falls out naturally (overrides, etc.).
 - `writable` Foo is a subclass of `readable` Foo
 - `immutable` Foo is also a subclass of `readable` Foo
 - `readable` object is the “top type” of the system

Permissions on Methods and References

```
class Person
{
    string m_name;

    public void SetName(string name) { ... }

    public string GetName() readable
    {

    }
}

readable Person p1 = ...;
```


Permissions and Generics

```
/// Deeply mutable object graph
```

```
List<Person> list1 = new List<Person>();  
list1.Add(new Person());  
list1[0].SetName(...);
```

```
/// Deeply readable: list or contents can't be mutated in any way
```

```
readable List<Person> list2 = list1;  
list2.Add(new Person());           // error CS7000: Calling 'List<T>.Add(T)' requires 'writable' permissions; you have 'readable'  
list2[0].SetName(...);             // error CS7000: Calling 'Person.SetName(string)' requires 'writable' permissions; you have 'readable'
```

```
/// Shallow readable: list contents can't changed but elements can
```

```
readable List<writable Person> list3 = list1;
```

Immutability

- Immutable types, but any reference to any type can also be immutable
- Simple to create immutable objects

```
immutable Person person = new Person() { Name = "Someone", Age = 42 };  
immutable List<Person> people = new List<Person>(  
    new[] { person, ... });
```

*Person and List<T> required **no** extra code to enable this scenario*

- Midori uses this to ensure all statics are immutable; as a result, statics can be evaluated and frozen at compile time

```
static immutable List<int> s_list = new List<int>(…) { … };
```

Error Model

Using static type systems to improve software quality

Motivation

Conflating Coding and Recoverable Errors

```
int Parse(string s)
{
    if (s == null) {
        throw new ArgumentException();
    }

    if (s == "One") {
        return 1;
    }
    else {
        throw new Exception();
    }
}

int Method()
{
    try {
        string s = GetString();
        return Parse(s);
    }
    catch (Exception) {
        return 0;
    }
}
```

Motivation

```
int Parse(string s)
{
    if (s == null) {
        throw new ArgumentNullException();
    }

    if (s == "One") {
        return 1;
    }
    else {
        throw new Exception();
    }
}

int Method()
{
    try {
        string s = GetString();
        return Parse(s);
    }
    catch (Exception) {
        return 0;
    }
}
```

Conflating Coding and Recoverable Errors

Which statement is throwing?

Error Philosophy

- **Solution:** Use the type-system to enforce a rigorous error model
- Fail fast on coding errors
 - Null dereferences, Array out-of-bounds
 - Overflows (checked arithmetic is the default)
 - OOM, Stack overflows
 - Contract failures
- Minimize recoverable failures
 - Usually a bug farm – not well tested
 - 90% of exceptions in .NET represent coding errors; see above
 - But they are still supported; e.g., parsers, reactive evaluations, I/O
- All error handling has explicit control flow, caller participates
- Static checking warns developers about dead code (catches with no throws)

Contracts and Assertions

- Contracts are fail-fast, subtyping-friendly, and side-effect-free

```
virtual string Format(int x)
    requires x > 0
    ensures return != null
{
    if (s == 1) {
        return "One";
    }
    else {
        return "More than one";
    }
}
```

```
override string Format(int x)
    requires base
    ensures base
{
    ...
    Debug.Assert(IsValid());
    ...
}

private bool IsValid() readable
{
    ...
}
```

Recoverable Errors: Exceptions

- Methods cannot throw by default
 - Predictability: >90% of our methods are guaranteed not to throw
 - Better code quality: Omit EH tables, more aggressive code motion, no branches (return codes)
- If a method can throw, callers and callee must opt-in

```
int Parse(string s)
{
    if (s == "One") {
        return 1;
    }
    else {
        throw new Exception(); // error
    }
}
```

```
int Method()
{
    string s = GetString();
    return Parse(s);
}
```


Wrap Up

...And Much More

- Many more features, in addition to nuances of the prior ones
 - Async model, similar to awaits but with lightweight linked stacks
 - Value type delegates
 - Fast interface pointers, improved dispatch to that of virtual calls
 - Primitive and readonly structs
 - Immutable types
 - Ref returns and ref locals
 - Typed exceptions
 - Lightweight enumeration
 - Copy Constructors
 - And a whole System C# / Midori Framework, inspired by .NET
 - See <http://mdk/> for a full list
- Roslyn-based IDE experience
 - Syntax highlighting, IntelliSense, refactoring, etc ...

Async Model

- Methods declare their ability to introduce interleaving.

```
awaits Client Connect(IPv4Address address)
{
    Socket socket = await Socket.Connect(address);
    return new Client(socket);
}
```

- Illegal to call awaiting code from non-awaiting code; spawn a task instead.

```
AsyncFactory factory = ...;
var result = factory.Run(() => await Connect(address));

await AsyncFactory.WaitAll(
    () => await Connect(address1)
    () => await Connect(address2)
);
```

- Implemented lightweight coroutines with linked stacks.
 - Zero-alloc 'awaits' methods.
 - Does not use state machines

Multiple Features in Action

```
public delegate int Comp<T>(readable T x, readable T y) readable;

public class List<T>
{
    public void Sort(Comparer<T>& comparer);
    public void Sort(Comp<T>& comparer);
}

public abstract immutable class Comparer<T>
{
    public void Compare(readable T x, readable T y);
}

// ...
List<Foo> foos = ...;

// No allocation!
// No side effects!
foos.Sort((x, y) => ... );
```

Future Plans

- Move towards open sourcing the language and compiler
 - Finish the language
 - Finish the port to Roslyn
 - Build on C# efforts to open source, message it right
 - Online community already expressed excitement (C# MVPs, D designers, Mono)
 - Continued collaboration with C# Team, feature transfer, design reviews
- Shipping in the box during Threshold
 - Conceptually Redhawk V2, we will ship ≥ 5 components in the box
 - Ahead of time compilation integrated with Windows tooling ecosystem
 - System C# can already target CLR, but loses most of our code quality advantage
 - For Threshold, it's mostly "us" writing the code; after, other OSG developers
- Looking forward to community feedback

Q & A

Yes, we are hiring